第5章 成员及成员指针

◆5.1 实例成员指针

- 运算符.*和->*均为双目运算符，优先级均为第14级，结合性自左向右。
- .*的左操作数为类的实例(对象)，右操作数为指向实例成员的指针。
- ->*的左操作数为对象指针，右操作数为指向该对象实例成员的指针。
- 实例成员指针是指向实例成员的指针，可分为实例数据成员指针和实例函数成员指针。
- 实例成员指针必须直接或间接同.*或->*左边的实例(对象)结合，以便访问该对象的实例数据成员或函数成员。
- 构造函数不能被显式调用，故不能有指向构造函数的实例成员指针。

第5章 成员及成员指针

◆5.1 实例成员指针

- 实例成员指针是成员相对于对象首地址的偏移，不是真正的代表地址的指针。
- 实例成员指针不能移动：
  - 数据成员的大小及类型不一定相同，移动后指向的内存可能是某个成员的一部分，或者跨越两个(或以上)成员的内存；
  - 即使移动前后指向的成员的类型正好相同，这两个成员的访问权限也有可能不同，移动后可能出现越权访问问题。
- 实例成员指针不能转换类型，否则便可以通过类型转换，间接实现实例成员指针移动。（参考课本P119例5.2）

第5章 成员及成员指针

◆5.2 const、volatile 和 mutable

- const 只读，volatile 易变，mutable 机动。
- const 和 volatile 可以定义变量、类的数据成员、函数成员及普通函数的参数和返回类型。
- mutable 只能用来定义类的实例数据成员。
- 含const实例数据成员的类必须定义构造函数（如果const实例数据成员没有设定缺省值），且数据成员必须在构造函数参数表之后，函数体之前初始化。
- 含 volatile、mutable 数据成员的类则不一定需要定义构造函数。

（参考课本P120例5.3）

第5章 成员及成员指针

◆5.2 const、volatile和mutable

- 普通函数成员参数表后出现const或volatile，修饰this指向的对象。出现const表示this指向的对象(其非静态数据成员)不能被函数修改，但可以修改this指向对象的非只读类型的静态数据成员。
- 构造或析构函数的this不能被说明为 const 或 volatile (即要构造或析构的对象应该能被修改，且状态要稳定不易变)。
- 对隐含参数的修饰还会会影响函数成员的重载：
- 普通对象应调用参数表后不带 const 和 volatile 的函数成员；
- const 和 volatile 对象应分别调用参数表后出现const和volatile的函数成员，否则编译程序会对函数调用发出警告。

（参考课本P121例5.4）

第5章 成员及成员指针

◆5.2 const、volatile 和 mutable

- 函数成员参数表后出现volatile，常表示调用该函数成员的对象是挥发对象，这通常意味着存在并发执行的进程。
- C++编译程序几乎都支持编写并发进程，编译时不对挥发对象作任何访问优化，即不利用寄存器存放中间计算结果，而是直接访问对象内存以便获得对象的最新值。
- 函数成员参数表后出现 const 时，不能修改调用对象的非静态数据成员，但如果该数据成员的存储类为 mutable，则该数据成员就可以被修改。
- mutable 用于说明实例数据成员，mutable 不能与 const、static 连用，但可以与 volatile 连用。

第5章 成员及成员指针

◆5.2 const、volatile和mutable

- 有址引用变量(&)只是被引用对象的别名，被引用对象自己负责构造和析构，该引用变量(逻辑上不分配内存的实体)不必构造和析构。
- 无址引用变量(&&)常用来引用缓存中的常量对象，该引用变量(逻辑上不分配缓存的实体)不必构造和析构。无址引用变量可为左值，但若同时用const定义则为传统右值。
- 如果A类型的有址引用变量r引用了new生成的(一定有址的）对象x，则应使用delete &r析构x，同时释放其所占内存。
- r.~A( )仅析构x而不释放其所占内存(由new分配)，造成内存泄漏。应该用delete &r;
- 引用变量必须在定义的同时初始化，引用参数则在调用函数时初始化。有址传统左值引用变量和参数必须用同类型的左值表达式初始化。

第5章 成员及成员指针

◆5.2 const、volatile和mutable

- mutable 仅用于说明实例数据成员为机动成员，不能用于静态数据成员的。
- 所谓机动是指在整个对象为只读状态时，其每个成员理论上都是不可写的，但若某个成员是mutable成员，则该成员在此状态是可写的。
- 例如，产品对象的信息在查询时应处于只读状态，但是其成员“查询次数”应在此状态可写，故可以定义为“机动”成员。
- 保留字mutable还可用于定义Lambda表达式的参数列表是否允许在Lambda的表达式内修改捕获的外部的参数列表的值。

（参考课本P123例5.6）

第5章 成员及成员指针

◆5.3 静态数据成员

- 静态数据成员是使用static说明或定义的类的数据成员。
- 静态数据成员通常在类的里面说明，在类的外面唯一定义一次。
- 静态数据成员一般用来描述类的总体信息，例如对象总个数。
- 实例数据成员可以定义默认值，但非const静态数据成员不能定义默认值。
- 静态数据成员在类中初始化只能定义为 inline static、const static、const inline 类型 (保留字顺序可变）。
- 静态数据成员不管是否用inline、const说明，在所有代码文件只有一个副本。
- 函数中的局部类不能定义静态数据成员，容易造成生命期矛盾。
- 静态数据成员不能定义为位段成员。

第5章 成员及成员指针

◆5.3 静态数据成员

例5.9函数局部类不能定义静态数据成员，全局类可以定义 inline 或 const 静态数据成员。

```
int  x = 3;
union  S { //定义局部类T
   const  static int b = 0;          //全局类中可用const定义同时初始化静态成员,必须用常量
   inline  static int c = x;         //全局类可用inline定义同时初始化静态成员,可用任意表达式
   inline  const static int d = x;  //可用任意表达式
};

void f(void) {
   class  T {     //定义函数中的局部类T
      int  d;     //static int d;  //错误：函数中的局部类不能定义静态数据成员
   };
   T a;           //局部自动变量 a
   static T s;    //局部静态变量 s
}
void main( ) {  f( ); f( );  }   //第一个函数调用f( )返回后 a.d ⇔ s.d ⇔ T::d 产生生成矛盾
```

第5章 成员及成员指针

◆5.4 静态函数成员

- 静态函数成员通常在类里以 static 说明或定义，它没有this参数。
- 有this的构造和析构函数、虚函数及纯虚函数都不能定义为静态函数成员。
- 静态函数成员一般用来访问类的总体信息，例如对象总个数。
- 静态函数成员可以重载、内联、定义默认值参数。
- 静态函数成员同实例成员的继承、访问规则没有没有太大区别。
- 静态函数成员的参数表后不能出现 const、volatile、const volatile 等修饰符。
- 静态函数成员的返回类型可以同时使用 inline、const、volatile 等修饰。

（参考课本P129例5.13）

第5章 成员及成员指针

◆5.5 静态成员指针

- 静态成员指针是指向类的静态成员的指针，包括静态数据成员指针和静态函数成员指针。
- 静态数据成员的存储单元为该类所有的对象共享，因此，通过该指针修改成员的值时会影响到所有对象该成员的值。
- 静态数据成员除了具有访问权限外，同普通变量没有本质区别；静态成员指针则和普通指针没有任何区别。
- 变量、数据成员、普通函数和函数成员的参数和返回值都可以定义成静态成员指针。

（参考课本P130例5.14）

第5章 成员及成员指针

◆5.5 静态成员指针

- 静态成员指针与普通成员指针有很大区别。静态成员指针存放成员地址，普通成员指针存放成员偏移；静态成员指针可以移动，普通成员指针不能移动；静态成员指针可以强制转换类型，普通成员指针不能强制转换类型。

```
struct A {
   int a, *b;
   int A::*u;   int A::*A::*x;
   int A::**y; int *A::*z;
   static int c, A::*d;
} z;
int A::c = 0;
int A::*A::d = &A::a;
void main(void) {
   int  i, A::**m;
   z.a = 5;   z.u = &A::a;   i = z.*z.u;
   z.x = &A::u;    i = z.*(z.*z.x);
   m = &A::d;
   m = &z.u;       i = z.**m;
   z.y = &z.u;     i = z.**z.y;
   z.b = &z.a;
   z.z = &A::b;    i = *(z.*z.z);
}
```

Problem：
假设 A x, y，对x和y执行main()中相同的程序，x、y里的成员变量哪些是相等的?

第5章 成员及成员指针

◆5.6 联合的成员指针

- 函数中局部类不能定义静态数据成员，故函数中的局部联合也不能定义。
- 全局类中的联合或全局联合可以定义静态数据成员。
- 静态数据成员指针一般指向全局类中的联合或全局联合的静态数据成员。
- 联合可以定义实例和静态函数成员，故也可以定义实例和静态函数成员指针。
- 联合的实例数据成员共享内存，因此，指向这些实例数据成员的指针存储的偏移量值实际上是相同的。

第12章 类型解析、转换与推导

◆12.1 隐式与显式类型转换

- 简单类型字节数：sizeof(bool) ≤ sizeof(char) ≤ sizeof(short)≤ sizeof(int) ≤ sizeof (long) ≤ sizeof(float) ≤ sizeof(double) ≤ sizeof(long double) 。
- 字节数少的类型向字节数多的类型转换时，一般不会引起数据的精度损失。
- 无风险的转换由编译程序自动完成，这种不提示程序员的自动转换也称为隐式类型转换。
- 隐式转换的基本方式: (1)非浮点类型字节少的向字节数多的转换;(2)非浮点类型有符号数向无符号数转换;(3)运算时整数向double类型的转换。
- 默认时，bool、char、short和int的运算按int类型进行，所有浮点常量及浮点数的运算按double类型进行。
- 赋值或调用时参数传递的类型相容，是指可以隐式转换，包括父子类的相容。

（参考课本P251例12.2）

第12章 类型解析、转换与推导

◆12.1 隐式与显式类型转换

- 可以设置VS2019给出最严格的编程检查：例如任何警告都报错等等。
- 有关类型转换若有警告，则应修改为强制类型转换即显式类型转换。
- 强制类型转换引起的问题由程序员负责。

```
char  u = 'a';          //编译时可计算，无截断，不报警
char  v = 'a' + 1;      // 编译时计算 'a' + 1 的值， 没有超过char的范围，不报警
char  v = 'a' + 100;    // 'a' + 100 超过char的范围（v=-59），报警，不截断
char  w = 300;          // 300超过char的范围（w=44），报警，截断
int     x = 2;          //x占用的字节数比char和short类型多，不报警
char  y = x;            //编译时不可计算，可能截断，报错
short z = x;            //编译时不可计算，可能截断，报错
```

第12章 类型解析、转换与推导

◆12.1 隐式与显式类型转换

- 一般简单类型之间的强制类型转换的结果为右值。
- 如果对可写变量进行同类型的左值引用转换，则转换结果为左值。
- 只读的简单类型变量如果转换为可写左值，并不能修改其值 (受到页面保护机制的保护）。

```
int x = 0;
(short)x = 2;       //报错：转换后((short) x)为传统右值，故不能出现在等号的左边
(int)x = 7;         //VS2019报错：传统右值不能出现在等号的左边。
(int &)x = 8;       //正确：x=8，用的不是最基本的简单类型，而是引用类型int &
const  int y = 9;
(int &)y = 10;      //y的结果仍然为9，全局变量如此赋值可引起程序异常（内存页面保护）
```

（参考课本P251例12.5）

第12章　类型解析、转换与推导

◆12.1 隐式与显式类型转换

●对于类的只读数据成员，如果转换为可写左值，可以修改其值。

●目前操作系统并不支持分层保护机制，无法在对象层和数据成员层提供不同类型的保护。【例12.4】

```
struct T {
    int  x = 0;
    const int y = 0;
    int q( ) {
        *(int *)&y = y + 1;
        return y;
    }
};
```

```
void main( ) {
    T m;
    const T n;
    int x = m.q( );   // 1
    x = m.q( );       // 2
    // x = n.q( );    // error
}
```

第12章　类型解析、转换与推导

◆12.2 cast系列类型转换

●static_cast同C语言的强制类型转换用法基本相同，不能从源类型中去除const和volitale属性，不做多态相关的检查。

●const_cast同C语言的强制类型转换用法基本相同，能去除或增加源类型的const和volitale属性。

●dynamic_cast将子类对象转换为父类对象时无须子类多态，而将基类对象转换为派生类对象时要求基类多态。

●reinterpret_cast主要用于名字同指针或引用类型之间的转换，以及指针与足够大的整数类型之间的转换。

第12章　类型解析、转换与推导

◆static_cast——静态转换

●使用格式为“static_cast<T> (expr)”，用于将数值表达式expr的源类型转换为T目标类型。

●目标类型不能包含存储位置类修饰符，如 static、extern、auto、register 等。

●static_cast 仅在编译时静态检查源类型能否转换为T类型，运行时不做动态类型检查。

●static_cast 不能去除源类型的const或volatile。即不能将指向const或volatile实体的指针(或引用)转换为指向非const或volatile实体的指针(或引用)。

（参考课本P256例12.6）

第12章　类型解析、转换与推导

◆const_cast——只读转换

●const_cast 的使用格式为 “const_cast<T>(左值表达式)”。

●修改类型的 const 和 volatile 属性，<T> 只能为指针、引用或指向对象成员的指针（可以带 const 和 volatile）。

●不能用 const_cast 将无址常量、位段访问、无址返回值转换为有址引用。

（参考课本P258例12.7）

第12章　类型解析、转换与推导

◆dynamic_cast——动态转换

- dynamic_cast 在运行时转换：派生类转换为基类、基类转换为派生类。
- dynamic_cast 主要用来解决将基类转换为派生类时的安全问题。
- 格式：dynamic_cast<T> (expr)
  类型T是类的引用、类的指针 或者 void *，
  expr的类型 必须是 类的对象 或者是 类的引用或指针。
- dynamic_cast 转换时不能去除expr源类型中的 const 和volitale 属性。
- 有址引用和无址引用之间不能相互转换。
- 将基类转换为派生类时，基类必须包含虚函数或纯虚函数。

第12章　类型解析、转换与推导

- dynamic_cast 主要用途

  将基类A的指针(引用)a转换成派生类B的指针(引用)b时，如果a所指的对象实际上不属于派生类B，则转换结果为NULL。

- 为什么dynamic_cast要求基类必须有虚函数？

  从dynamic_cast主要用途知道，dynamic_cast需要知道类的继承关系。而类的继承关系，可以从虚函数表中分析出来。因此，需要基类具有虚函数。

- 如果基类不含虚函数，则在编译时 dynamic_cast 就会报错

（参考课本P262例12.11）

第12章　类型解析、转换与推导

◆reinterpret_cast——重释转换

- reinterpret_cast <T> (expr)，将表达式expr转换成不同性质的其他类型T。T类型不能是实例数据成员指针。T 可以是指针、引用、或其他与expr完全不同的类型。
- 将指针转换为足够大的整数，整数类型必须够存储一个地址。X86和X64的指针大小不同，X86使用int类型即可。
- 当T为使用&或&&定义的引用类型时，expr 必须是一个左值表达式。
- 左值引用和右值引用可以相互转换。

（参考课本P264例12.13）

第12章　类型解析、转换与推导

◆12.3 类型转换实例

- C++的父类指针（或引用）可以直接指向（或引用）子类对象，但是通过父类指针（或引用）只能调用父类定义的成员函数。
- 武断或盲目地向下转换，然后访问派生类或子类成员，会引起一系列安全问题: (1)成员访问越界(如父类无子类的成员); (2)函数不存在(如父类无子类函数) 。
- 关键字 typeid 可以获得对象的真实类型标识：有 ==、!=、before、raw_name、hash_code 等函数。
- typeid使用格式：(1) typeid (类型表达式)； (2) typeid (数值表达式)。
- typeid的返回结果是 const type_info & 类型，在使用 typeid 之前先 #include <typeinfo>。（参考课本P266例12.14）

第12章　类型解析、转换与推导

◆12.3 类型转换实例

●保留字explicit只能用于定义构造函数或类型转换实例函数，
explicit 定义的实例函数成员必须显式调用。

```
class COMPLEX {
   double r, v;
public:
   explicit COMPLEX(double r1 = 0, double v1 = 0)
   {  r = r1; v = v1;  }
   COMPLEX operator+(const COMPLEX &c) const
   {  return COMPLEX(r + c.r, v + c.v);  };
   explicit operator double( ) { return r; }
} m(2, 3);
```

未用explicit定义前:
(1) double d = m 等价于 d = m.operator double( )
(2) m+2.0 等价于 m + COMPLEX(2.0, 0.0)

使用explicit定义后:
(1) 不能定义 d = m;
(2) 不能用 m + 2.0 相加。

只能:

```
double d = m.operator double( );
COMPLEX a = m + COMPLEX(2.0, 0.0);
```

第12章　类型解析、转换与推导

◆12.4 自动类型推导

●保留字 auto 在C++中用于类型推导。

●可用于推导变量、各种函数的返回值、以及类中用const定义的静态数据成员的类型。

●使用auto推导时，被推导实体不能出现类型说明，但是可以出现存储可变特性 const、voilatile 和存储位置特性如 static、register。

（参考课本P271例12.18）

第12章　类型解析、转换与推导

◆12.4 自动类型推导

●保留字 auto 可以推导与数组和函数相关的类型。

●数组名代表整个数组类型，函数名代表该函数的指针。

●auto 数组：将 数组类型 的第1维 (最低维) 推导为指针，后面的维类型不变。

●无论被推导变量前面有无 *，auto将函数名和数组名解释为指针。

第12章　类型解析、转换与推导

```
#include <iostream>
#include <typeinfo>
using namespace std;
int   a[3][4][5];   //可理解为 int (*a)[4][5]
auto  b = &a;       //int (*b)[3][4][5]  (a的类型是int [3][4][5], 取其地址)
auto  *bp = &a;     //int (*bp)[3][4][5] (auto c⇔auto *cp)
auto  c = a;        //int (*c)[4][5] (a的类型是int [3][4][5], 将其第1维推导为指针)
auto  *cp = a;      //int (*cp)[4][5] (auto b⇔auto *bp, b的类型是数组, 所以解释为指针)
auto  d = a[1];     //int (*d)[5] (a[?]的类型是int [4][5], 将其第1维推导为指针)
auto  *dp = a[1];         //int (*dp)[5]  (auto d⇔auto *dp)
auto  e = a[1][2];        //int *e  (a[?][?]的类型是int [5], 将其第1维推导为指针)
auto  *ep = a[1][2];      //int *ep (auto e⇔auto *ep)
auto  f = a[1][1][1];     //int g
auto  *fp = a[1][1][1];   //error (a[?][?][?]的类型是int, 不是数组, 因此与auto *fp不等价)
auto  h(int x) { return x; };  //int h(int)
auto  g = printf;               //int (*g)(const char *, …)
```

第12章　类型解析、转换与推导

```
int main( ) {
   auto m = { 1, 2, 3 };      //int m[3] = { 1, 2, 3 }
   auto n = new auto(1);   //int *n = new int(1)
   auto p = h;                //int (*p)(int)  (函数名总是解释为指针)
   auto *q = h;               //int (*q)(int)  (h是函数, 因此 auto p ⇔ auto *q)
   (*p)(4);                   //调用 b(4)
   (*q)(5);                   //调用 b(5)

   cout << typeid(a).name();      //int [3][4][5]  (注意 auto  c = a 中c的类型)
   cout << typeid(a[1]).name();  //int [4][5]      (注意 auto  d = a[1] 中d的类型)
   cout << typeid(d).name();     //int (*)[5]
   cout << typeid(p).name();     //int (_cdecl *)(int)
   cout << sizeof(c);             //4
}
```

第12章　类型解析、转换与推导

◆12.4　自动类型推导

- 关键字 decltype 用来提取表达式的类型。
- 凡是需要类型的地方均可出现 decltype。
- 可用于变量、成员、参数、返回类型的定义
  以及 new、sizeof、异常列表、强制类型转换。
- 可用于构成新的类型表达式。

第12章　类型解析、转换与推导

```
int  a[10][20];
auto r = new decltype(a);    //int (*r)[20] = new int[10][20];
decltype(a) *p = &a;          //int (*p)[10][20]
decltype(&a[0]) h(decltype(a) x, int y) { return x; };   //int (*h(int x[10][20], int))[20]
//不能定义decltype(a) h(decltype(a) x, int y, int z);     //C++的函数不能返回2维数组
void sort(double *a, unsigned N, bool (*g)(double, double)) {
   for (int x = 0; x < N - 1; x++)
      for (int y = x + 1; y < N; y++)
         if ((*g)(a[x], a[y])) { double t = a[x]; a[x] = a[y]; a[y] = t; }
}
auto f(double x, double y) throw(const char *)   //bool (*f)(double, double)
{  return x > y;  };
```

第12章　类型解析、转换与推导

```
auto g = [ ](int x)->int { return x; };        //匿名对象被g创建, 已经知道类型
decltype(g) (*q)[10];                          //正确：表达式g的类型已被计算出来
decltype([ ](int x)->int{ return x; }) *q;   //错误：匿名对象未被创建, 不知道类型

void main() {
   double a[5];
   decltype(a) *r;   //a的类型为double [5]，r的为double (*)[5]
   a[0] = 1;  a[1] = 5;  a[2] = 3;  a[3] = 2;  a[4] = 4;
   sort(a, sizeof(decltype(a)) / sizeof(double), f);
}
```

第12章　类型解析、转换与推导

◆12.5　Lambda表达式

- lambda表达式是一个匿名函数，该函数实现了一个匿名类，匿名类中主要包括匿名的构造函数和( )运算符重载函数 operator( )。
- lambda表达式的函数体就是operator( )。调用 lambda表达式实际上就是调用 ( )运算符的重载函数 operator( )。
- operator() 是 const 的，即 operator( )(…) const，因此不能修改匿名类内的任何实例数据成员。
- 定义一个lambda表达式，则创建一个匿名类，同时创建该匿名类的一个对象。

第12章　类型解析、转换与推导

lambda表达式形式:

[capture list] (parameter list) -> return type { function body }

capture list: 捕获列表，用于获得lambda函数体外变量的值（lambda表达式根据这些捕获到的变量创建匿名类的同名实例成员变量）。捕获可以分为按值捕获和按引用捕获。非局部变量，如静态变量、全局变量等不需要捕获，直接使用。

parameter list: 参数列表（调用时传入的参数），可以省略。从C++14开始，支持默认参数。

return type: 返回值类型。可以省略，这种情况下根据lambda函数体中的return语句推断出返回类型，如果函数体中没有return，则返回类型为void。

function body: 函数体（即 ( )运算符的重载函数 operator( )(…) 的函数体）。

第12章　类型解析、转换与推导

Lambda表达式的调用方式:

```
auto f = [ ](int x)->int { return x * x; };  //创建一个匿名类, 同时创建对象f.
int x = f(10);  //等价:  int x = f.operator()(10)  ()运算符的调用方式??
```

捕捉变量: 捕捉lambda函数体外变量的值（lambda表达式根据这些捕获到的变量创建匿名类的实例成员变量）。

[ ] 不捕获任何变量。

[&] 以引用方式捕获所有变量。

[=] 用值的方式捕获所有变量。

[varName] 以值方式捕获变量varName。

[this] 捕获所在类的this指针。

第12章　类型解析、转换与推导

Lambda表达式的本质:

```
int a = 1;
int main() {
    static int b = 2;
    int  m = 3, n = 4;
    char *s = new char [10] {'a', 'b', 'c', 0};
    auto f = [m, &n, s](int x)->char * {
        s[0] += m+n+x+a+b;
        //m++;  //错:  匿名类的实例数据成员m是const的
        n++;   //对:  匿名类的实例数据成员n是引用变量,
               //      可以修改所指向的内存单元
        a++;   //对:  全局变量a不是匿名类的数据成员
        b++;   //对:  静态变量b不是匿名类的数据成员
        return s;
    }; //创建匿名类及其对象f
    f(5)[0] = '1';       //等价:  f.operator()(5)[0] = '1'
    std::cout << s;    //1bc
    f.operator()(6);   //等价:  f(6)
    std::cout << s;    //Dbc
}
```

Lambda表达式的解释:

为了方便解释，下面用 A 表示匿名类的名称。

调用 f(…) ⇔ f.operator()(…)

```
class A {
    int  m, &n;
    char *s;
    A(int m, int &n, char *s): m(m), n(n), s(s) { }
    char *operator()(int x) const {
        (A::s)[0] += A::m + A::n + x + a + b;
        //A::m++;  //错, 不能改变A::m
        A::n++;
        ::a++;
        b++;   //main::b++
        return A::s;
    }
} f;
```

第12章　类型解析、转换与推导

Lambda表达式的调用机制：

- 定义Lambda表达式及其对象时，将创建一个匿名类，同时创建一个该对象。
- 每次调用 Lambda表达式，都是利用该对象去调用（ ）运算符重载函数operator( )，即：对象.operator( )(…)，也可写成：对象(…)。
- operator()的属性是const的，因此 operator()不能修改匿名类中的非引用类型的实例成员变量。但通过 将 Lambda 表达式修改为 mutable 属性，使得匿名类中的所有实例成员变量都具备 mutable 属性，这样 operator( ) 可以修改匿名类中所有的实例成员变量。

第12章　类型解析、转换与推导

Lambda表达式的调用机制解释：

```
int main() {
    static int a = 1;
    int  m = 2;
    auto f = [m](int x) mutable -> int {
          m += a + x;
          return m;
    }; //创建匿名类及其对象f
    int  i = f(0);   //i = 3 （ f.operator()(0) ）
    int  j = f(0);   //j = 4
    printf("%d, %d, %d \n", m, i, j); //2, 3, 4
}
```

对Lambda表达式的解释：

为了方便解释，下面用 A 表示匿名类的名称。

f (…) ⇔ f.operator()(…)

```
class A {
    mutable int  m;
    A(int m): m(m) { }
    int  operator( )(int x) const {
          A::m += a + x;   // A::m += main::a + x
          return A::m;
    }
} f;
```

第12章　类型解析、转换与推导

```
#include <iostream>
int a = 1;
int main( ) {
    static int x = 3;
    int y = 4;
    int z = 5;
    auto f = [y, &z](int v) -> int {
        //y++;   //错：f是const, 不能修改匿名类的成员变量y
        z++;     //对：z是引用, 可以修改引用所指的变量
        return a+x+y+z+v;
    }; //创建一个匿名类及其对象f
    auto g = [=](int v) -> int {
        //z++;   //错：  g是const, 不能修改匿名类的成员变量z
        return a+x+y+z+v;
    }; //创建一个匿名类及其对象g
    auto h = [=](int v) mutable -> int {
        y++;   //对：  mutable int y
        z++;   //对：  mutable int z
        return a+x+y+z+v;
    }; //创建一个匿名类及其对象h
    int z1 = f(100);   // z1 = ?
    int z2 = g(100);   // z2 = ?
    int z3 = h(100);   // z3 = ?
}
```

z1 = 114
z2 = 113
z3 = 115

如果将 int z1 = f(100) 移到 auto g = [=] 之前，z1、z2、z3 的值又是多少?

z1 = 114
z2 = 114
z3 = 116

第12章　类型解析、转换与推导

- **Lambda表达式的匿名类与普通匿名类的区别**
  - 普通的匿名类可以生成多个有名字的对象，Lambda表达式的匿名类只能产生一个有名字的对象（这个对象是在定义Lambda表达式时创建的）；
  - 普通匿名类的实例成员函数有对象的 this 指针，Lambda表达式匿名类的实例成员函数没有匿名类对象的 this 指针.
- **可以用函数指针指向捕获列表为空的Lambda表达式的 ( ) 函数**

```
auto f = [ ](int x)->int { return x * x; };
auto g = [ y ](int x)->int { return x + y; };
int (*p)(int) = f;   //对，p 指向匿名类的函数operator()(int x)
int  z = p(10);      //z = 100
int (*q)(int) = g;   //错，g的捕获列表不为空
```

(1) 这里 f 和 g 的类型是2个匿名类，而 p 和 q 是2个普通函数的指针。

(2) p的类型不能auto，即不能 auto (*p)(int) = f，因为 f的类型是匿名类，编译器不会将 f.operator()(int x) 的返回值类型推断给 p。

（参考课本P274例12.21）